GymnasiumWildeshausen

Tschöpe, Erik

(Name, Vorname)

Eigene Programmiersprache?

WebAssembly macht's möglich!

(Titel der Facharbeit)

Erstellt im Rahmen des Seminarfachs:

Die Zukunft des Digitalen

Schuljahr 2025/2026

GymnasiumWildeshausen

Name der/s Schülerin/s: Erik Tschöpe

Titel der Arbeit: Eigene Programmiersprache ? WebAssembly machts möglich!

Name der Fachlehrerkraft: Herr Gewald

Ausgabetermin des Themas: 08.01.2026

Abgabetermin der Arbeit: 19.02.2026

Erik Tschöpe

(Unterschrift der Schülerin/des Schülers)

Die vorliegende Facharbeit wurde am 19.02.2026 eingereicht.

Note: _____ / _____ Punkte

_____

Unterschrift der Fachlehrerin/ des Fachlehrers

Inhaltsverzeichnis

1 Einleitung

Unterschiedliche Programmiersprachen unterscheiden sich in Syntax, Funktionsumfang und Einsatzbereich. So ist es ein Traum vieler Entwickler, sich eine eigene Programmiersprache zu erschaffen, die perfekt an die eigenen Bedürfnisse angepasst ist.

Trotzdem bleibt die Entwicklung einer eigenen Programmiersprache für viele Entwickler ein fernes Ziel, da sie häufig mit einem hohen technischen Aufwand verbunden ist und viel Erfahrung erfordert. Ein wesentlicher Grund dafür liegt im Bau eines sogenannten Compilers.

Ein Compiler ist ein Programm, das Quellcode in eine für den Computer ausführbare Form übersetzt. Der komplexeste Teil ist dabei das Backend, das für die Erzeugung von plattformspezifischem Maschinencode verantwortlich ist. Unterschiedliche Prozessorarchitekturen und Betriebssysteme erfordern jeweils eigene Lösungen, was den Entwicklungsaufwand stark erhöht. Aus diesem Grund werden Compiler in der Regel von größeren Entwicklerteams oder Unternehmen realisiert und nur selten von einzelnen Amateurentwicklern.

Mit der Einführung von WebAssembly (WASM) im Jahr 2017 wurde ein neuer Ansatz vorgestellt, der genau an dieser Stelle ansetzt. WebAssembly stellt ein standardisiertes, plattformunabhängiges Ausführungsformat dar [1], das von modernen Webbrowsern und zunehmend auch außerhalb des Webs unterstützt wird. Statt direkt Maschinencode für eine bestimmte Architektur zu erzeugen, können Compiler WebAssembly als gemeinsames Ziel verwenden. Dadurch werden viele hardware- und betriebssystemspezifische Details abstrahiert.

Diese Entwicklung wirft die Frage auf, ob WebAssembly den Einstieg in den Compilerbau grundlegend erleichtert. Insbesondere stellt sich die Frage, ob es durch WebAssembly erstmals realistisch wird, dass auch Amateurentwickler eigene, funktionsfähige Compiler entwickeln können.

Aus diesem Zusammenhang ergibt sich die Leitfrage dieser Facharbeit: **Inwiefern erleichtert WebAssembly den Bau eigener Compiler für Amateurentwickler?**

Zur Beantwortung dieser Frage werden zunächst die grundlegenden Konzepte des Compilerbaus sowie die Funktionsweise von WebAssembly erläutert. Anschließend wird in einem praktischen Selbstversuch ein einfacher Mini-

Compiler für eine eigens definierte Programmiersprache entwickelt, der WebAssembly-Bytecode generiert. Auf Basis dieser Erfahrungen werden die Chancen und Grenzen von WebAssembly als Zielplattform für Hobby-Compiler bewertet.

Im weiteren Verlauf wird untersucht, inwiefern WebAssembly den Einstieg in den Compilerbau erleichtert. Im Fokus stehen dabei die plattformunabhängige Ausführung und die Abstraktion von Hardwaredetails, durch die sich die Implementierung stärker auf Sprach- und Compilerlogik konzentrieren kann.

2 Vorwissen: Compiler und WebAssembly

Dieses Kapitel führt in die Grundkonzepte des Compilerbaus ein und erläutert die Architektur eines Compilers. Der Fokus liegt auf dem Verständnis, warum WebAssembly als Zielplattform eine vielversprechende Vereinfachung für Hobby-Compiler darstellt.

2.1 Was ist ein Compiler?

Ein Compiler ist ein Programm, welches Programmcode in eine andere, für Computer verständliche Form übersetzt. Dabei ist es ganz egal, ob es Binärcode für eine bestimmte Prozessorarchitektur, Bytecode für eine virtuelle Maschine oder eine Zwischenrepräsentation für die Weiterverarbeitung ist. In Abgrenzung zu einem Interpreter führt ein Compiler den Code nicht direkt aus, sondern übersetzt ihn nur und führt dabei optional Optimierungen durch. Kompilierte Programme laufen dadurch in der Regel schneller als interpretierte Programme, da die Übersetzung bereits vor der Ausführung stattfindet und Optimierungen vorgenommen werden können. Zusätzlich erleichtern moderne Compiler den Entwicklern das Leben, indem sie häufige Fehler schon beim Übersetzen des Quellcodes finden und verständliche Fehlermeldungen ausgeben, während Interpreter Fehler erst zur Laufzeit sichtbar werden, was die Fehlersuche erschwert [2]. Wenn der Begriff „Compiler“ fällt, ist selten nur der reine Übersetzungsvorgang gemeint, sondern oft die gesamte Toolchain, die auch Assembler und Linker umfasst, um aus Quellcode eine ausführbare Datei zu erzeugen [3].

2.2 Aufbau eines Compilers (Frontend, Backend)

Ein Compiler ist grundlegend in mehrere Teile unterteilt, die jeweils klar abgegrenzte Aufgaben übernehmen: Lexing (Erzeugung von Token aus Quelltext), Parsing (Aufbau eines abstrakten Syntaxbaums, AST), semanti-

sche Analyse (Typprüfung, Namensauflösung, Scope- und Fehlerprüfung), eine Zwischenrepräsentation und Optimierungsphase (IR-Transformationen, konstante Auswertung, Dead-Code-Elimination) sowie das Backend (Code- bzw. Bytecode-Generierung, z.B. für WebAssembly). Diese Modularität erleichtert Entwicklung, Testbarkeit und Wiederverwendbarkeit der einzelnen Komponenten. Zusätzlich bietet diese Struktur die Möglichkeit, verschiedene Frontends (für unterschiedliche Sprachen) mit demselben Backend zu kombinieren, was die Flexibilität erhöht. Moderne Compiler sind genau entlang solcher Schritte aufgebaut [4].

2.3 Einführung in WebAssembly

WebAssembly (WASM) ist ein binäres, plattformunabhängiges Ausführungsformat, das ursprünglich für die Ausführung in Webbrowsern entwickelt wurde, aber inzwischen auch außerhalb des Webs, z.B. auf Servern oder in Tools, genutzt werden kann. Es zielt darauf ab, eine nahe an nativer Geschwindigkeit liegende Ausführung zu ermöglichen, während es gleichzeitig portabel und sicher bleibt. WASM-Module bestehen aus Funktionen, Speicher, Tabellen sowie Import- und Exportdefinitionen. Die Ausführung erfolgt in einer Sandbox-Umgebung, wobei der Zugriff auf Systemfunktionen über Host-Imports erfolgt. WASM wird von vielen Sprachen unterstützt, darunter C/C++, Rust und AssemblyScript, die über Compiler-Toolchains in WASM übersetzt werden können. Für den Compilerbau bietet WASM eine einheitliche Zielplattform, wodurch viele plattformspezifische Details im Backend entfallen. Unter anderem deswegen ist WASM besonders attraktiv für Hobby-Compiler, da es die Komplexität der Codegenerierung reduziert und den Fokus auf die Sprachlogik und -semantik ermöglicht [5].

Das Grundprinzip, nach dem WASM arbeitet, ist die Stack-Maschine, bei der Instruktionen primär auf einem Operand-Stack operieren. Zum Beispiel nimmt die `add`-Instruktion die obersten zwei Werte vom Stack, addiert sie und legt das Ergebnis wieder auf den Stack. Dies ermöglicht eine einfache und effiziente Ausführung von Anweisungen, da keine expliziten Register oder Speicheradressen benötigt werden [6]; [1].

WebAssembly wird in Modulen verpackt, die Funktionen, Speicher, Tabellen sowie Import- und Exportdefinitionen enthalten. Ein valides WASM-Modul muss bestimmte Regeln erfüllen, damit es von der Laufzeitumgebung akzeptiert wird. Dazu gehören ein klarer Aufbau des Moduls, die Konsistenz von

Funktionssignaturen, die Korrektheit des Kontrollflusses und die Gültigkeit referenzierter Indizes. Der Validator prüft diese Regeln vor der Ausführung, und bei Verstoß wird das Modul nicht instanziiert [7]; [1].

Hier ist ein Beispiel für Quellcode in unserer eigenen Sprache, der eine einfache Addition durchführt, sowie das daraus generierte WebAssembly-Textformat (WAT). Das Beispiel zeigt, wie eine Funktion `add` definiert wird, die zwei Ganzzahlen addiert, und eine `main`-Funktion, die diese Addition ausführt und das Ergebnis über eine Host-Import-Funktion `print` ausgibt.

```
fn add(a, b) -> Int {
    return a + b;
}

fn main(){
    print(add(7, 5));
    return;
}
```

Codebeispiel 1: Programmbeispiel: Addition in eigener Sprache (Quelle: add.eres)

Aus diesem Beispielprogramm wird folgendes WAT generiert, das die gleiche Logik in WebAssembly-Textformat darstellt. Es definiert die Funktion `add`, die zwei `i64`-Parameter entgegennimmt und deren Summe zurückgibt, sowie die `main`-Funktion, die `add(7, 5)` aufruft und das Ergebnis mit der importierten Funktion `print_i64` ausgibt.

```
(module
  (type (;0;) (func (param i64)))  // Funktion mit 1 Parameter, kein
Rückgabewert
  (type (;1;) (func (param i64 i64) (result i64)))  // Funktion mit 2 Parametern
und Rückgabe
  (type (;2;) (func))  // Funktion ohne Parameter, ohne Rückgabe
  (import "env" "print_i64" (func (;0;) (type 0)))  // Import: print_i64 aus env
  (export "add" (func 1))  // Export: Funktion 1 heißt add
  (export "main" (func 2))  // Export: Funktion 2 heißt main

  (func (;1;) (type 1) (param i64 i64) (result i64)  // Funktion add(a,b) -> i64
    local.get 0  // lade Parameter a
    local.get 1  // lade Parameter b
    i64.add  // addiere a + b
    return  // Ergebnis zurückgeben
    i64.const 0  // (Default-Return, falls kein return)
    return
  )

  (func (;2;) (type 2)  // Funktion main()
    i64.const 7  // konstante 7 auf den Stack
    i64.const 5  // konstante 5 auf den Stack
    call 1  // rufe add(7,5) auf
    call 0  // rufe print_i64(result) auf
    return  // Ende
  )
)
```

Codebeispiel 2: Generiertes WAT zum Additionsbeispiel (aus `add.eres`)

3 Selbstversuch

Die Konzepte des Compilerbaus und die Funktionsweise von WebAssembly wurden nun theoretisch erläutert. Um die praktische Umsetzbarkeit dieser Konzepte zu überprüfen, wird im folgenden Abschnitt ein eigener Mini-Compiler entwickelt. Dieser Compiler soll eine eigens definierte, minimalistische Programmiersprache in WebAssembly-Bytecode übersetzen. Dabei wird die gesamte Pipeline von der Quelltexteingabe über die Tokenisierung, das Parsing, die semantische Analyse bis hin zur Codegenerierung und Ausführung durchlaufen. Ziel ist es, nicht nur die technischen Schritte zu demonstrieren, sondern auch konkrete Artefakte wie die erzeugten Tokens, den abstrakten Syntaxbaum (AST), das generierte WAT und die Laufzeitausgabe zu präsentieren und nachvollziehbar zu machen.

3.1 Methodik des Selbstversuchs

Methodisch handelt es sich um eine prototypische Implementierung mit bewusst begrenztem Sprachumfang. Geprüft wird jede Stufe der Übersetzung mit passenden Zwischenergebnissen (Token-Stream, AST und WAT) sowie

anschließend die End-to-End-Ausführung über Wasmtime. Die Auswertung erfolgt durch den Vergleich von erwartetem und tatsächlichem Verhalten anhand eigener Testprogramme; der Fokus liegt dabei auf Nachvollziehbarkeit der Pipeline, nicht auf Produktionsreife oder umfassender Optimierung. Um die Reproduzierbarkeit zu sichern, wurden die Testläufe mit festen Eingabedateien und konsistenten CLI-Optionen durchgeführt und die Zwischenausgaben jeweils dokumentiert.

3.2 Funktionsumfang der eigenen Programmiersprache

Diese Sprache ist bewusst minimalistisch gehalten, um den Fokus auf die Kernkonzepte des Compilerbaus zu legen. Sie unterstützt nur einen Datentyp (`Int`), der intern als 64-Bit-Ganzzahl (`i64`) umgesetzt wird. Es gibt keine globalen Variablen, sondern nur Funktionen, die lokale Variablen über `let` -Statements definieren können. Kontrollstrukturen umfassen `if/else` und `while`, während Ausdrücke Literale, Variablen, Binäroperationen und Funktionsaufrufe erlauben. Vergleichsoperatoren ermöglichen einfache Bedingungen. Rückgabetypen sind optional, und eine Host-Funktion `print` ermöglicht die Ausgabe von Werten.

```
#[derive(Debug)]
pub enum Stmt { // Mögliche Anweisungen in der Sprache
    Let { name: String, value: Expr },
    Return(Option<Expr>),
    Expr(Expr),
    If { cond: Expr, then_block: Vec<Stmt>, else_block: Vec<Stmt> },
    While { cond: Expr, body: Vec<Stmt> },
}

#[derive(Debug)]
pub struct FunctionDecl { // Hält Signatur und Funktionskörper zusammen
    pub name: String,
    pub params: Vec<String>,
    pub body: Vec<Stmt>,
    pub return_type: Option<Type>,
}
```

Codebeispiel 3: AST-Datenstrukturen (Quelle: src/ast.rs)

3.3 Lexer / Scanner

Der Lexer liest den Quelltext Zeichen für Zeichen und gruppiert sie in sinnvolle Einheiten, sogenannte Tokens. Er erkennt Schlüsselwörter wie `fn`, `let`, `if`, `else`, `while` und `return`, die eine spezielle Bedeutung haben. Außerdem identifiziert er Literale (z.B. Ganzzahlen) und Identifier (z.B. Funktions- oder Variablennamen). Operatoren und Trennzeichen werden ebenfalls als eigene

Token klassifiziert. Bei der Verarbeitung des Quelltexts muss der Lexer auch Fehler erkennen, z.B. wenn ein unerwartetes Zeichen auftaucht oder eine Zahl ungültig formatiert ist [8]; [9]. Die verschiedenen Token-Typen werden in einem Enum `TokenKind` modelliert, das die verschiedenen Kategorien von Tokens abdeckt, einschließlich Schlüsselwörtern, Literalen, Operatoren und Fehlern.

```
pub enum TokenKind {
    // Schlüsselwörter
    Let, Return, If, While,
    // Inhalte
    Ident(String), Int(i64),
    // Operatoren / Trenner
    Plus, Minus, Star, Slash, Equal,
    LParen, RParen, LBrace, RBrace, Semicolon,
    // Ende / Fehler
    EOF,
    Error,
}
```

Codebeispiel 4: Token-Typen (Quelle: src/token.rs)

Beim Hauptlauf des Lexers wird der Quelltext zeichenweise durchlaufen. Zunächst werden alle Whitespace-Zeichen übersprungen, da sie für die Syntax keine Bedeutung haben. Sobald ein nicht-Whitespace-Zeichen gefunden wird, entscheidet der Lexer, ob es sich um den Beginn eines Identifiers, einer Zahl oder eines Operators handelt. Ein Identifier oder Schlüsselwort beginnt mit einem Buchstaben oder Unterstrich, gefolgt von alphanumerischen Zeichen oder Unterstrichen. Eine Zahl besteht ausschließlich aus Ziffern. Operatoren und Trennzeichen werden direkt erkannt. Sobald ein Token vollständig erkannt ist (z.B. wenn kein weiteres Zeichen mehr zum aktuellen Token passt), wird es ausgegeben.

```rust
fn next_token(chars: &mut std::iter::Peekable<std::str::Chars<'_>>) -> TokenKind
{
    while let Some(&c) = chars.peek() {
        if c.is_whitespace() {
            chars.next(); // Whitespace überspringen
            continue;
        }

        if c.is_ascii_alphabetic() || c == '_' {
            return lex_ident(chars); // Identifier oder Schlüsselwort
        }
        if c.is_ascii_digit() {
            return lex_number(chars); // Zahl
        }

        chars.next();
        return match c {
            '+' => TokenKind::Plus, // Operatoren und Trennzeichen
            '-' => TokenKind::Minus,
            ';' => TokenKind::Semicolon,
            _ => TokenKind::Error,
        };
    }

    TokenKind::EOF // Ende des Quelltexts
}
```

Codebeispiel 5: Vereinfachter Lexer-Hauptlauf (Quelle: src/lexer.rs)

Bei der Erkennung von Identifiers wird erst am Ende entschieden, ob es sich um ein Schlüsselwort handelt, indem die gesammelte Zeichenfolge mit bekannten Schlüsselwörtern verglichen wird. Wenn ein unbekanntes Zeichen auftaucht, erzeugt der Lexer sofort einen Fehler-Token, ohne weiter zu parsen.

```rust
fn keyword_or_ident(text: String) -> TokenKind {
    match text.as_str() {
        "let" => TokenKind::Let, // Deklaration von Variablen
        "return" => TokenKind::Return, // Rückgabe von Funktionen
        "if" => TokenKind::If, // Bedingte Anweisungen
        "while" => TokenKind::While, // Schleifen
        _ => TokenKind::Ident(text), // Identifier
    }
}
```

Codebeispiel 6: Identifier-Lexing (Quelle: src/lexer.rs)

3.4 Parser

Der Parser nimmt die vom Lexer erzeugte Tokenliste und baut daraus einen abstrakten Syntaxbaum (AST, Abstract Syntax Tree) auf, der die hierarchische Struktur des Programms widerspiegelt. Die in diesem Selbstversuch verwendete Strategie nennt sich rekursiver Abstieg, wobei durch Rekursion

die hierarchische Natur der Sprache direkt in der Parserlogik abgebildet wird ([10]).

Unterschieden wird zwischen Statements, die vollständige Anweisungen darstellen (z.B. `let x = 5;` oder `return x;`), und Expressions, die einen Wert liefern (z.B. `x + 2` oder `add(7, 5)`). In einem Statement wie `let y = x + 2;` ist das gesamte Konstrukt ein Statement, während `x + 2` die Expression ist, die den Wert für die Variable `y` liefert. Ebenso ist in `print(add(7, 5));` das gesamte Konstrukt ein Statement, während `add(7, 5)` die Expression ist, deren Ergebnis an die `print`-Funktion übergeben wird.

3.4.1 Rekursiver Abstieg

So können beispielsweise Blöcke, die aus einer Sequenz von Statements bestehen, einfach durch eine Funktion `parse_block` umgesetzt werden, die so lange Statements parst, bis sie das schließende `}` findet. Der Einstiegspunkt ist die Funktion `parse_program`, die alle Funktionen im Quelltext sammelt, bis sie das End-Token (`EOF`) erreicht. Jede Funktion wird durch `fn name(params) -> Int { ... }` definiert, wobei der Rückgabetyp optional ist. Blöcke werden als Sequenzen von Statements in `{ ... }` dargestellt.

```
fn parse_program(&mut self) -> Result<Program, ParseError> {
    let mut functions = Vec::new(); // Funktionsliste.

    while self.peek().kind != TokenKind::EOF { // Bis Dateiende.
        let func = self.parse_function()?; // Eine Funktion parsen.
        functions.push(func); // Hinzufügen.
    }

    Ok(Program { functions }) // AST zurück.
}
```

Codebeispiel 7: Parser-Einstieg (*`parse_program`*) (vereinfacht, Quelle: src/parser.rs)

```
fn parse_block(&mut self) -> Result<Vec<Stmt>, ParseError> {
    self.expect(TokenKind::LBrace)?; // `{` erwartet.
    let mut statements = Vec::new(); // Statements sammeln.

    while self.peek().kind != TokenKind::RBrace { // Bis `}`.
        let stmt = self.parse_stmt()?; // Statement parsen.
        statements.push(stmt); // Speichern.
    }

    self.expect(TokenKind::RBrace)?; // `}` erwartet.
    Ok(statements) // Block zurück.
}
```

Codebeispiel 8: Block-Parser (*`parse_block`*) (vereinfacht, Quelle: src/parser.rs)

3.4.2 Operator-Präzedenz

Um die korrekte Bindung von Operatoren zu gewährleisten, wird ein Präzedenzsystem implementiert. Dabei wird jedem Operator eine Präzedenzstufe zugeordnet, die bestimmt, in welcher Reihenfolge die Operatoren ausgewertet werden. Zum Beispiel bindet `*` stärker als `+`, sodass `1 + 2 * 3` als `1 + (2 * 3)` interpretiert wird. Dies wird durch eine Prioritätstabelle gesteuert, die die Präzedenz der Operatoren definiert. [10]

```rust
fn precedence(kind: &TokenKind) -> Option<u8> {
    match kind {
        TokenKind::Star | TokenKind::Slash => Some(20), // Hoch.
        TokenKind::Plus | TokenKind::Minus => Some(10), // Niedrig.
        _ => None, // Kein Operator.
    }
}
```

Codebeispiel 9: Operator-Präzedenz (vereinfacht, Quelle: src/parser.rs)

Wenn der Parser ein erwartetes Token nicht findet, bricht er sofort mit einem Fehler (`Err`) ab und meldet, was erwartet wurde und was tatsächlich gefunden wurde.

```rust
fn parse_function(&mut self) -> Result<FunctionDecl, ParseError> {
    self.expect(TokenKind::Fn)?; // `fn`.
    let name = self.expect_ident()?; // Name.

    self.expect(TokenKind::LParen)?; // `(`.
    let params = self.parse_params()?; // Parameter.
    self.expect(TokenKind::RParen)?; // `)`.

    let return_type = if self.peek().kind == TokenKind::Arrow {
        self.bump(); // `->`.
        self.expect(TokenKind::IntType)?; // Typ.
        Some(Type::Int) // Mit Rückgabetyp.
    } else {
        None // Ohne Rückgabetyp.
    };

    let body = self.parse_block()?; // Funktionsblock.

    Ok(FunctionDecl {
        name, // Name
        params, // Parameter
        body, // Statements
        return_type, // Rückgabetyp
    })
}
```

Codebeispiel 10: Funktionsparser (Quelle: src/parser.rs)

3.5 Codegen

Der Codegenerator nimmt den AST und übersetzt ihn in eine eigene Zwischenrepräsentation (IR, Intermediate Representation), die aus einer linearen Folge von Instruktionen besteht. Diese IR abstrahiert von den Details der WASM-Generierung und ermöglicht eine klarere Trennung zwischen der Logik der Codeerzeugung und den spezifischen Anforderungen des WASM-Formats. Zusätzlich erleichtert die IR spätere Optimierungen, da Transformationen nicht mehr direkt auf der Quellsyntax arbeiten müssen. Der Einsatz einer IR entspricht der Praxis großer Compiler-Infrastrukturen, wie z.B. LLVM IR als zentrale Zwischenschicht [11]. In diesem Fall ist die IR eine einfache Aufzählung von Instruktionen, die direkt auf den WASM-Stack operieren. Zum Beispiel wird ein `if`-Statement in der IR durch eine Sequenz von Instruktionen dargestellt, die die Bedingung evaluieren, dann eine `If`-Instruktion mit einem Blocktyp für den `then`-Teil und optional eine `Else`-Instruktion für den `else`-Teil enthält. Ähnlich wird eine `while`-Schleife durch einen `Block` und einen `Loop` mit entsprechenden Sprunginstruktionen (`BrIf`, `Br`) modelliert.

```
pub enum IrInstruction{
    I64Const(i64), // Laden einer Konstanten
    I64Eqz, // Vergleich: ist 0?
    BrIf(u32), // Bedingter Sprung (z.B. für `if` oder `while`)
    Br(u32), // Unbedingter Sprung (z.B. für `break` in `while`)
    LocalSet(u32), // Setzen einer lokalen Variable
    LocalGet(u32), // Laden einer lokalen Variable
    Call(u32), // Funktionsaufruf
    If(BlockType), // Wenn-Block
    Else, // Sonst-Block
    Block(BlockType), // Block
    Loop(BlockType), // Schleife
    Drop, // Wert vom Stack entfernen
    Return, // Rückgabe
    End, // Ende

    // Arithmetik
    I64Add, // Addition
    I64Sub, // Subtraktion
    I64Mul, // Multiplikation
    I64DivS, // Division
    I64Eq, // Gleichheit
    I64LtS, // Kleiner als
    I64GtS, // Größer als
    I64ExtendI32S, // Erweiterung von i32 nach i64, nötig für Vergleiche mit 0
    I32Eqz, // Vergleich: ist 0?
}
```

Codebeispiel 11: IR-Instruktionen (Quelle: src/codegen/ir.rs)

Die IR wird letztendlich in echte WASM-Bytecode-Instruktionen umgewandelt, wobei die `wasm_encoder`-Bibliothek verwendet wird, um Module, Funktionen, Imports und Exports zu definieren.

```rust
pub fn emit_function(&mut self, func: &FunctionDecl) {
    let mut gen = FuncGen {
        locals: Vec::new(), // Lokale Variablen (inkl. Parameter)
        local_map: HashMap::new(), // Mapping von Namen zu lokalen Indizes
        instructions: Vec::new(), // IR-Instruktionen
        has_return: func.return_type.is_some(), // Rückgabetyp vorhanden?
    };

    // Parameter werden auf lokale Indizes gemappt
    for (i, name) in func.params.iter().enumerate() {
        gen.local_map.insert(name.clone(), i as u32);
    }

    // Statements -> IR-Instruktionen
    for stmt in &func.body {
        emit_stmt(stmt, &mut gen, &self.func_indices);
    }
}
```

Codebeispiel 12: Funktions-Emission (Quelle: src/codegen/module.rs)

Die Host-Funktion `print` wird als Import unter dem Namen `env.print_i64` bereitgestellt, damit sie im generierten WASM-Modul aufgerufen werden kann und die Ausgabe von Ganzzahlen über die Konsole ermöglicht.

3.6 Ausführung und Beispiel

Für diesen Selbstversuch wird der generierte WASM-Bytecode lokal mit der `wasmtime`-Laufzeit ausgeführt, anstatt ihn in einem Browser zu verwenden, auch wenn dies theoretisch möglich wäre. `wasmtime` ermöglicht das Laden von WASM-Modulen, die Instanziierung von Funktionen und den Aufruf von exportierten Funktionen wie `main`. Die Host-Funktion `print_i64` wird in Rust bereitgestellt und als Import in das WASM-Modul eingebunden, damit die `print(...)`-Funktion im generierten Code funktioniert. Der Ablauf umfasst die Umwandlung des Quelltexts in Tokens durch den eigenen Lexer, den Aufbau eines AST durch den Parser und die Generierung von WASM-Bytes durch den Codegenerator. Anschließend erfolgt die Ausführung mit Wasmtime [12].

```rust
pub fn run_wasm_bytes(bytes: &[u8], args: Vec<i64>) -> Result<Option<i64>,
String> {
    let engine = Engine::default();
    let module = wasmtime::Module::from_binary(&engine, bytes)
        .map_err(|e| format!("module compile error: {}", e))?;

    let mut store = Store::new(&engine, ());

    // Host-Funktion für print(...)
    let print_func = wasmtime::Func::wrap(&mut store, |v: i64| {
        println!("{}", v);
    });

    // Instanziierung mit Importen (z.B. `env.print_i64`)
    let instance = Instance::new(&mut store, &module, &[print_func.into()])
        .map_err(|e| format!("instance error: {}", e))?;

    // Aufruf der exportierten `main`-Funktion
    let func = instance
        .get_func(&mut store, "main")
        .ok_or_else(|| "function `main` not found".to_string())?;
    func.call(&mut store, &params, &mut results_buf)
        .map_err(|e| format!("runtime error: {}", e))?;
}
```

Codebeispiel 13: WASM-Ausführung mit Wasmtime (Quelle: src/runner.rs)

Die Ausführung eines Programms in dieser Programmiersprache startet also immer mit der `main`-Funktion, die als Einstiegspunkt dient. Alle anderen Funktionen müssen von `main` oder von anderen Funktionen aufgerufen werden, damit sie ausgeführt werden. Wenn `main` nicht definiert ist oder nicht exportiert wird, schlägt die Instanziierung des WASM-Moduls fehl, da der Einstiegspunkt fehlt.

3.6.1 Beispielprogramm: Fakultät

Hier ein Beispielprogramm, das die Fakultät einer Zahl berechnet. Es zeigt die Verwendung von Funktionen, Rekursion und die Ausgabe über die `print` -Funktion. Das Programm definiert eine Funktion `fact`, die die Fakultät berechnet, und eine `main`-Funktion, die `fact(10)` aufruft und das Ergebnis ausgibt.

```
fn fact(n) -> Int {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}

fn main(){
    print(fact(10));
    return;
}
```

Codebeispiel 14: Programmbeispiel: Fakultät (Quelle: factorial.eres)

Wenn man dieses Programm mit `cargo run -- factorial.eres` ausführt, wird die Ausgabe `3628800` (die Fakultät von 10) in der Konsole angezeigt, was die korrekte Funktion des Compilers und der Laufzeit bestätigt.

3.7 Eigener Beitrag und verwendete Tools

Die Eigenleistung umfasst die vollständige Implementierung eines Mini-Compilers von der Quelltextanalyse (Lexer, Parser) über die Codegenerierung bis zur Ausführung des generierten WASM-Codes. Die Sprachsyntax wurde von mir definiert, und alle Komponenten des Compilers wurden eigenständig entwickelt. Zusätzlich habe ich Testfälle erstellt, um die Funktionalität von Lexer, Parser und der Laufzeit zu validieren. Die verwendeten Tools umfassen die Programmiersprache Rust für die Implementierung, die `wasm-encoder`-Bibliothek für die Generierung von WASM-Bytecode, die `wasmtime`-Laufzeit für die Ausführung des generierten Codes, Git für Versionsverwaltung sowie Typst für die Erstellung dieser Facharbeit. Das zugehörige Projekt-Repository ist unter github.com/justEres/facharbeit verfügbar. KI-Unterstützung wurde für die Strukturierung der Arbeit und das Brainstorming von Ideen genutzt, jedoch nur wenig für die eigentliche Code-Implementierung.

3.8 Testwerkzeuge und CLI-Nutzung

Um die Entwicklung zu vereinfachen und die Funktionalität zu demonstrieren, wurden verschiedene Testwerkzeuge und CLI-Optionen implementiert. Die wichtigsten Befehle sind:

Token-Ausgabe (zeigt die Arbeit des Lexers):

```
$ cargo run -- add.eres --print-tokens
Tokens:
[
  Token { kind: Fn, ... },
  Token { kind: Ident("add"), ... },
  Token { kind: LParen, ... },
  ...
  Token { kind: EOF, ... }
]
```

Codebeispiel 15: CLI: Token-Stream prüfen (`--print-tokens`)

AST-Ausgabe (zeigt die vom Parser erzeugte Baumstruktur):

```
$ cargo run -- add.eres --print-ast
AST:
Program {
  functions: [
    FunctionDecl { name: "add", ... },
    FunctionDecl { name: "main", ... },
  ],
}
```

Codebeispiel 16: CLI: AST prüfen (`--print-ast`)

WAT-Ausgabe (zeigt die Codegen-Ausgabe als WebAssembly-Text):

```
$ cargo run -- add.eres --print-wat
Generated WAT:
(module
  (import "env" "print_i64" ...)
  (export "add" (func 1))
  (export "main" (func 2))
  ...
)
```

Codebeispiel 17: CLI: WAT prüfen (`--print-wat`)

4 Fazit

Um die Leitfrage zu beantworten: WebAssembly erleichtert den Bau eigener Compiler für Amateurentwickler erheblich, da es ein einheitliches Ziel bietet und viele plattformspezifische Details im Backend abstrahiert. Allerdings bleibt die Entwicklung eines Compilers eine komplexe Aufgabe, insbesondere im Frontend (Lexing, Parsing, semantische Analyse). Die Abhängigkeit von Host-Imports und der Sandbox-Umgebung von WASM kann ebenfalls Einschränkungen mit sich bringen. Insgesamt ist WASM eine vielversprechende Plattform für Hobby-Compiler, aber es erfordert dennoch ein gewisses Maß an technischem Verständnis und Aufwand.

Mein persönliches Fazit ist, dass die Entwicklung dieses Mini-Compilers eine äußerst lehrreiche Erfahrung war. Sie hat mir geholfen, die theoretischen Konzepte des Compilerbaus in die Praxis umzusetzen und die Herausforderungen zu verstehen, die mit der Erstellung eines Compilers verbunden sind. Obwohl das Projekt eher ein Proof of Concept als ein produktionsreifer Compiler ist, hat es mir wertvolle Einblicke gegeben und meine Fähigkeiten im Bereich Compilerentwicklung deutlich verbessert.

Die größte Herausforderung lag in der Implementierung der Parserlogik, insbesondere bei der Handhabung von Operatorpräzedenz und der Fehlerbehandlung. Auch die Codegenerierung für WASM war komplex, da ich mich mit den Details der WASM-Instruktionen und der Modulstruktur auseinandersetzen musste. Das Tooling rund um die Entwicklung eines Compilers, einschließlich Testen und Debuggen, stellte ebenfalls eine Herausforderung dar.

Um das Projekt weiterzuführen, wären Verbesserungen in der Fehlerdiagnostik wünschenswert, z.B. durch genauere Fehlermeldungen mit Zeilen- und Spaltenangaben sowie die Möglichkeit zur Fehlererholung. Ein weiterer Schritt wäre die Implementierung eines Typcheckers, um statische Typfehler zu erkennen. Schließlich könnte der Sprachumfang erweitert werden, um weitere Datentypen, Kontrollstrukturen oder Funktionen zu unterstützen.

5 Quellen

Ausschließlich Online-Quellen, da keine gedruckten Bücher verwendet wurden. Alle Quellen sind frei zugänglich und wurden zum Zeitpunkt des Zugriffs überprüft. Die Auswahl der Quellen basiert auf ihrer Relevanz für die Themen Lexer, Parser, Codegen, WASM und Compilerbau im Allgemeinen.

1. Bytecode Alliance: Hello, world! - Wasmtime. Verfügbar unter: https://docs.wasmtime.dev/examples-hello-world.html. Zugriff am 12.02.2026.

2. Bytecode Alliance: wasmtime crate documentation. Verfügbar unter: https://docs.rs/wasmtime/latest/wasmtime/. Zugriff am 12.02.2026.

3. GNU Project: Overall Options (Using the GNU Compiler Collection). Verfügbar unter: https://gcc.gnu.org/onlinedocs/gcc-14.1.0/gcc/Overall-Options.html. Zugriff am 12.02.2026.

4. IBM: Was ist ein Compiler?. Verfügbar unter: https://www.ibm.com/de-de/think/topics/compiler. Zugriff am 16.02.2026.

5. LLVM Project: LLVM Language Reference Manual. Verfügbar unter: https://llvm.org/docs/LangRef.html. Zugriff am 12.02.2026.

6. MDN Web Docs: WebAssembly concepts. Verfügbar unter: https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Concepts. Zugriff am 16.02.2026.

7. MDN Web Docs: Understanding WebAssembly text format. Verfügbar unter: https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Understanding_the_text_format. Zugriff am 16.02.2026.

8. Robert Nystrom: Parsing Expressions (Chapter 6) - Crafting Interpreters. Verfügbar unter: https://craftinginterpreters.com/parsing-expressions.html. Zugriff am 16.02.2026.

9. Robert Nystrom: Scanning (Chapter 4) - Crafting Interpreters. Verfügbar unter: https://craftinginterpreters.com/scanning.html. Zugriff am 16.02.2026.

10. Rust Compiler Development Guide: Overview of the compiler. Verfügbar unter: https://rustc-dev-guide.rust-lang.org/overview.html. Zugriff am 16.02.2026.

11. Rust Compiler Development Guide: Lexing and parsing. Verfügbar unter: https://rustc-dev-guide.rust-lang.org/the-parser.html. Zugriff am 16.02.2026.

12. W3C: WebAssembly Core Specification (W3C Working Draft). Verfügbar unter: https://www.w3.org/TR/2024/WD-wasm-core-2-20240217/. Zugriff am 12.02.2026.

13. WebAssembly Community Group: WebAssembly Core Specification. Verfügbar unter: https://webassembly.github.io/spec/core. Zugriff am 21.01.2026.

6 Einsatz von KI

Für die Facharbeit wurde KI als unterstützendes Hilfsmittel für Strukturierung, sprachliche Überarbeitung und Layout in Typst genutzt. Die inhaltliche Entwicklung des Selbstversuchs, der Compiler-Code und die technischen Entscheidungen wurden eigenständig erarbeitet.

GymnasiumWildeshausen

E R K L Ä R U N G

Hiermit erkläre ich, dass ich die vorliegende Facharbeit selbstständig und ohne fremde Hilfe verfasst und sämtliche hierfür zu Hilfe genommenen gedruckte sowie digitale Quellen im Literatur- bzw. Quellenverzeichnis angegeben und die aus diesen Quellen stammenden Zitate und Belegstellen für sinngemäß wiedergegebene Inhalte in meiner Seminarfacharbeit als solche kenntlich gemacht habe.

Wildeshausen , den 18.02.2026

(Ort) (Datum)

Erik Tschöpe

(Unterschrift der Schülerin/des Schülers)

7 Anhang: Quellcode

Quellcode-Übersicht

7.0.1 factorial.eres

```
// factorial example
fn fact(n) -> Int {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}

fn main(){
    print(fact(10));
    return;
}
```

7.0.2 src/ast.rs

```
#[derive(Debug)]
pub enum Expr {
    Int(i64),
    Local(String),
    Binary {
        op: BinOp,
        left: Box<Expr>,
        right: Box<Expr>,
    },
    Call {
        name: String,
        args: Vec<Expr>,
```

```
    },
}

#[derive(Debug)]
pub enum BinOp {
    Add,
    Sub,
    Mul,
    Div,
    Eq,
    NotEq,
    Lt,
    Le,
    Gt,
    Ge,
}

#[derive(Debug)]
pub enum Stmt {
    Let {
        name: String,
        value: Expr,
    },
    Return(Option<Expr>),
    Expr(Expr),
    If {
        cond: Expr,
        then_block: Vec<Stmt>,
        else_block: Vec<Stmt>,
    },
    While {
        cond: Expr,
        body: Vec<Stmt>,
    },
}

#[derive(Debug)]
pub struct FunctionDecl {
    pub name: String,
    pub params: Vec<String>,
    pub body: Vec<Stmt>,
    pub return_type: Option<Type>,
}

#[derive(Debug)]
pub enum Type {
    Int,
}

#[derive(Debug)]
pub struct Program {
    pub functions: Vec<FunctionDecl>,
}
```

7.0.3 src/lexer.rs

```
use std::str::Chars;

use crate::token::{Span, Token, TokenKind};

pub struct Lexer<'a> {
    chars: Chars<'a>,
    pos: usize,
}

impl<'a> Lexer<'a> {
    fn peek(&self) -> Option<char> {
        self.chars.clone().next()
    }

    fn bump(&mut self) -> Option<char> {
        let c = self.chars.next()?;
        self.pos += c.len_utf8();
        Some(c)
    }

    fn span_from(&self, start: usize) -> Span {
        Span {
            start,
            end: self.pos,
        }
    }

    pub fn next_token(&mut self) -> Result<Token, LexError> {
        self.skip_whitespace();

        let start = self.pos;

        let ch = match self.bump() {
            Some(c) => c,
            None => {
                return Ok(Token {
                    kind: TokenKind::EOF,
                    span: Span { start, end: start },
                });
            }
        };

        let kind = match ch {
            '(' => TokenKind::LParen,
            ')' => TokenKind::RParen,
            '{' => TokenKind::LBrace,
            '}' => TokenKind::RBrace,
            ';' => TokenKind::Semicolon,
            '+' => TokenKind::Plus,
            ':' => TokenKind::Colon,
            ',' => TokenKind::Comma,
            '>' => {
                if let Some('=') = self.peek() {
                    self.bump();
```

```
            TokenKind::GreaterEqual
        } else {
            TokenKind::Greater
        }
    }
    '<' => {
        if let Some('=') = self.peek() {
            self.bump();
            TokenKind::LessEqual
        } else {
            TokenKind::Less
        }
    }
    '-' => {
        if let Some('>') = self.peek() {
            self.bump();
            TokenKind::Arrow
        } else {
            TokenKind::Minus
        }
    }
    '*' => TokenKind::Star,
    '/' => {
        if let Some('/') = self.peek() {
            // Single-line comment
            while let Some(c) = self.peek() {
                if c == '\n' {
                    break;
                }
                self.bump();
            }
            return self.next_token();
        } else {
            TokenKind::Slash
        }
    }
    '%' => TokenKind::Percentage,
    '=' => {
        if let Some('=') = self.peek() {
            self.bump();
            TokenKind::EqualEqual
        } else {
            TokenKind::Equal
        }
    }
    '!' => {
        if let Some('=') = self.peek() {
            self.bump();
            TokenKind::NotEqual
        } else {
            return Err(LexError::UnexpectedChar {
                ch: '!',
                span: self.span_from(start),
            });
        }
    }
```

```
            c if c.is_ascii_digit() => self.lex_number(c)?,
            c if Lexer::is_ident_start(c) => self.lex_ident(c),

            _ => {
                return Err(LexError::UnexpectedChar {
                    ch,
                    span: self.span_from(start),
                });
            }
        };

        Ok(Token {
            kind,
            span: self.span_from(start),
        })
    }

    pub fn lex_number(&mut self, first_digit: char) -> Result<TokenKind, LexError>
{
        let start = self.pos - first_digit.len_utf8();
        let mut number_str = first_digit.to_string();

        while let Some(c) = self.peek() {
            if c.is_ascii_digit() {
                number_str.push(c);
                self.bump();
            } else {
                break;
            }
        }

        match number_str.parse::<i64>() {
            Ok(value) => Ok(TokenKind::Int(value)),
            Err(_) => Err(LexError::InvalidNumber {
                span: self.span_from(start),
            }),
        }
    }

    pub fn is_ident_start(c: char) -> bool {
        c.is_ascii_alphabetic() || c == '_'
    }

    pub fn lex_ident(&mut self, first_char: char) -> TokenKind {
        let mut ident_str = first_char.to_string();

        while let Some(c) = self.peek() {
            if c.is_ascii_alphanumeric() || c == '_' {
                ident_str.push(c);
                self.bump();
            } else {
                break;
            }
        }

        match ident_str.as_str() {
```

```
            "let" => TokenKind::Let,
            "fn" => TokenKind::Fn,
            "if" => TokenKind::If,
            "else" => TokenKind::Else,
            "while" => TokenKind::While,
            "return" => TokenKind::Return,
            "Int" => TokenKind::IntType,
            _ => TokenKind::Ident(ident_str),
        }
    }

    fn skip_whitespace(&mut self) {
        while let Some(c) = self.peek() {
            if c.is_whitespace() {
                self.bump();
            } else {
                break;
            }
        }
    }

    pub fn new(src: &'a str) -> Self {
        Lexer {
            chars: src.chars(),
            pos: 0,
        }
    }
}

pub fn lex_file(src: &str) -> Result<Vec<Token>, LexError> {
    let mut lexer = Lexer::new(src);
    let mut tokens = Vec::new();

    loop {
        let token = lexer.next_token()?;
        if token.kind == TokenKind::EOF {
            tokens.push(token);
            break;
        }
        tokens.push(token);
    }

    Ok(tokens)
}

#[derive(Debug)]
pub enum LexError {
    UnexpectedChar { ch: char, span: Span },
    InvalidNumber { span: Span },
}

pub fn report_lex_error(_src: &str, error: LexError) {
    match error {
        LexError::UnexpectedChar { ch, span } => {
            eprintln!(
                "LexError: Unexpected character '{}' at {}:{}",
```

```
                ch, span.start, span.end
            );
        }
        LexError::InvalidNumber { span } => {
            eprintln!("LexError: Invalid number at {}:{}", span.start, span.end);
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn lex_simple_tokens() {
        let src = "( ) { } ; + - * / % , : -> <= < >= > == != ->";
        // include arrow twice intentionally
        let tokens = lex_file(src).expect("lexing failed");
        // ensure EOF is last
        assert_eq!(tokens.last().unwrap().kind, TokenKind::EOF);
        // Check that some known tokens appear in order (spot check)
        let kinds: Vec<_> = tokens.iter().map(|t| &t.kind).collect();
        assert!(kinds.contains(&&TokenKind::LParen));
        assert!(kinds.contains(&&TokenKind::LessEqual));
        assert!(kinds.contains(&&TokenKind::GreaterEqual));
        assert!(kinds.contains(&&TokenKind::EqualEqual));
        assert!(kinds.contains(&&TokenKind::NotEqual));
    }
}
```

7.0.4 src/main.rs

```
use clap::Parser;

use crate::codegen::module::ModuleGen;

mod ast;
mod codegen;
mod lexer;
mod parser;
mod runner;
mod token;

#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
struct Args {
    /// Input source file
    input: String,

    /// Print tokens produced by the lexer
    #[arg(long, default_value_t = false)]
    print_tokens: bool,

    /// Print the parsed AST
    #[arg(long, default_value_t = false)]
```

```
    print_ast: bool,

    /// Print generated WAT before running
    #[arg(long, default_value_t = false)]
    print_wat: bool,
}

fn main() {
    let args = Args::parse();

    let src = std::fs::read_to_string(&args.input).expect("Failed to read input
file");

    let bytes = match lexer::lex_file(&src) {
        Ok(tokens) => {
            if args.print_tokens {
                println!("Tokens:\n{:#?}", tokens);
            }

            let mut parser = parser::Parser::new(&tokens);
            let ast = parser.parse_program().unwrap();

            if args.print_ast {
                println!("AST:\n{:#?}", ast);
            }

            let mut module_gen = ModuleGen::new();
            module_gen = module_gen.init_with_host_functions();

            // Declare first, then emit, so functions can reference each other.
            for func in &ast.functions {
                module_gen.declare_function(func);
            }
            for func in &ast.functions {
                module_gen.emit_function(func);
            }

            let bytes = module_gen.finish();

            // Determine the arity of main to prepare call arguments.
            let main_param_count = ast
                .functions
                .iter()
                .find(|f| f.name == "main")
                .map(|f| f.params.len())
                .unwrap_or(0);

            Some((bytes, args.print_wat, main_param_count))
        }
        Err(e) => {
            lexer::report_lex_error(&src, e);
            None
        }
    };

    if let Some((bytes, print_wat, param_count)) = bytes {
```

```
        let wat = wasmprinter::print_bytes(&bytes).unwrap();
        if print_wat {
            println!("Generated WAT:\n{}", wat);
        }

        // Prepare zero-initialized i64 args matching the function's param count.
        let args: Vec<i64> = vec![0; param_count];

        match runner::run_wasm_bytes(&bytes, args) {
            Ok(Some(result)) => println!("result of main function: {}", result),
            Ok(None) => println!("main returned no value"),
            Err(e) => eprintln!("Execution error: {}", e),
        }
    }
}
```

7.0.5 src/parser.rs

```
use crate::ast::*;
use crate::token::*;

#[derive(Debug)]
pub struct Parser<'a> {
    tokens: &'a [Token],
    pos: usize,
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::lexer::lex_file;

    #[test]
    fn parse_paren_comparison() {
        let src = "fn main(x, y) -> Int { if (x < y) { return 1; } else { return
0; } }";
        let tokens = lex_file(src).expect("lex");
        let mut p = Parser::new(&tokens);
        let program = p.parse_program().expect("parse");

        // one function
        assert_eq!(program.functions.len(), 1);
        let f = &program.functions[0];
        assert_eq!(f.body.len(), 1);

        match &f.body[0] {
            Stmt::If {
                cond,
                then_block,
                else_block,
            } => {
                match cond {
                    Expr::Binary { op, left, right } => {
                        match op {
                            BinOp::Lt => (),
```

```
                        _ => panic!("expected Lt op"),
                    }

                    match (&**left, &**right) {
                        (Expr::Local(l), Expr::Local(r)) => {
                            assert_eq!(l, "x");
                            assert_eq!(r, "y");
                        }
                        _ => panic!("expected locals in comparison"),
                    }
                }
                _ => panic!("expected binary cond"),
            }

            assert_eq!(then_block.len(), 1);
            assert_eq!(else_block.len(), 1);
        }
        _ => panic!("expected if statement"),
    }
}

#[test]
fn parse_mixed_precedence() {
    let src = "fn main(x, y) -> Int { let z = x + 1 < y; }";
    let tokens = lex_file(src).expect("lex");
    let mut p = Parser::new(&tokens);
    let program = p.parse_program().expect("parse");
    let f = &program.functions[0];

    match &f.body[0] {
        Stmt::Let { name, value } => {
            assert_eq!(name, "z");
            match value {
                Expr::Binary { op, left, right } => {
                    match op {
                        BinOp::Lt => (),
                        _ => panic!("expected Lt op"),
                    }
                    match &**left {
                        Expr::Binary {
                            op: lop,
                            left: lleft,
                            right: _lright,
                        } => {
                            match lop {
                                BinOp::Add => (),
                                _ => panic!("expected Add as left op"),
                            }
                            match &**lleft {
                                Expr::Local(n) => assert_eq!(n, "x"),
                                _ => panic!("expected local x"),
                            }
                        }
                        _ => panic!("expected binary left side"),
                    }
                    match &**right {
```

```
                            Expr::Local(n) => assert_eq!(n, "y"),
                            _ => panic!("expected local y"),
                        }
                    }
                    _ => panic!("expected binary value"),
                }
            }
            _ => panic!("expected let statement"),
        }
    }

    #[test]
    fn parse_parenthesized_complex() {
        let src = "fn main(a,b,c,d) -> Int { let r = (a + b) <= (c - d); }";
        let tokens = lex_file(src).expect("lex");
        let mut p = Parser::new(&tokens);
        let program = p.parse_program().expect("parse");
        let f = &program.functions[0];

        match &f.body[0] {
            Stmt::Let { name, value } => {
                assert_eq!(name, "r");
                match value {
                    Expr::Binary {
                        op,
                        left: _left,
                        right: _right,
                    } => match op {
                        BinOp::Le => (),
                        _ => panic!("expected Le op"),
                    },
                    _ => panic!("expected binary value"),
                }
            }
            _ => panic!("expected let statement"),
        }
    }
}

impl<'a> Parser<'a> {
    pub fn new(tokens: &'a [Token]) -> Parser<'a> {
        Parser { tokens, pos: 0 }
    }

    fn peek(&self) -> &Token {
        self.tokens.get(self.pos).unwrap()
    }

    fn bump(&mut self) -> Token {
        let tok = self.peek().clone();
        self.pos += 1;
        tok
    }

    fn expect(&mut self, kind: TokenKind) -> Result<Token, ParseError> {
        let tok = self.bump();
```

```
    if tok.kind == kind {
        Ok(tok)
    } else {
        Err(ParseError::UnexpectedToken {
            expected: kind.name(),
            found: tok,
        })
    }
}

pub fn parse_program(&mut self) -> Result<Program, ParseError> {
    let mut functions = Vec::new();

    while self.peek().kind != TokenKind::EOF {
        let func = self.parse_function()?;
        functions.push(func);
    }
    Ok(Program { functions })
}

fn parse_function(&mut self) -> Result<FunctionDecl, ParseError> {
    self.expect(TokenKind::Fn)?;

    let name = match self.bump().kind.clone() {
        TokenKind::Ident(s) => s,
        tok => {
            return Err(ParseError::UnexpectedToken {
                expected: "identifier".to_string(),
                found: Token {
                    kind: tok,
                    span: self.peek().span.clone(),
                },
            });
        }
    };

    self.expect(TokenKind::LParen)?;

    let mut params = Vec::new();
    if self.peek().kind != TokenKind::RParen {
        loop {
            match self.bump().kind.clone() {
                TokenKind::Ident(s) => params.push(s),
                _ => {
                    return Err(ParseError::UnexpectedToken {
                        expected: "parameter name".to_string(),
                        found: self.peek().clone(),
                    });
                }
            }

            if self.peek().kind == TokenKind::Comma {
                self.bump();
            } else {
                break;
            }
```

```
            }
        }

        self.expect(TokenKind::RParen)?;
        // Optional return type (`-> Int`).
        let return_type = if self.peek().kind == TokenKind::Arrow {
            self.bump();
            // Currently only Int is supported.
            self.expect(TokenKind::IntType)?;
            Some(crate::ast::Type::Int)
        } else {
            None
        };
        let body = self.parse_block()?;

        Ok(FunctionDecl {
            name,
            params,
            body,
            return_type,
        })
    }

    fn parse_block(&mut self) -> Result<Vec<Stmt>, ParseError> {
        self.expect(TokenKind::LBrace)?;
        let mut stmts = Vec::new();

        while self.peek().kind != TokenKind::RBrace {
            let stmt = self.parse_stmt()?;
            stmts.push(stmt);
        }

        self.expect(TokenKind::RBrace)?;
        Ok(stmts)
    }

    fn parse_stmt(&mut self) -> Result<Stmt, ParseError> {
        match self.peek().kind {
            TokenKind::Let => self.parse_let(),
            TokenKind::Return => self.parse_return(),
            TokenKind::If => self.parse_if(),
            TokenKind::While => self.parse_while(),
            _ => self.parse_expr_stmt(),
        }
    }

    fn parse_let(&mut self) -> Result<Stmt, ParseError> {
        self.expect(TokenKind::Let)?;

        let name = match self.bump().kind.clone() {
            TokenKind::Ident(s) => s,
            _ => {
                return Err(ParseError::UnexpectedToken {
                    expected: "identifier".to_string(),
                    found: self.peek().clone(),
                });
```

```
        }
    };

    self.expect(TokenKind::Equal)?;
    let value = self.parse_expr()?;
    self.expect(TokenKind::Semicolon)?;

    Ok(Stmt::Let { name, value })
}

fn parse_return(&mut self) -> Result<Stmt, ParseError> {
    self.expect(TokenKind::Return)?;

    // Allow `return;` with no expression. If present, parse an expression.
    if self.peek().kind == TokenKind::Semicolon {
        self.bump();
        Ok(Stmt::Return(None))
    } else {
        let expr = self.parse_expr()?;
        self.expect(TokenKind::Semicolon)?;
        Ok(Stmt::Return(Some(expr)))
    }
}

fn parse_if(&mut self) -> Result<Stmt, ParseError> {
    self.expect(TokenKind::If)?;

    let cond = self.parse_expr()?;

    let then_block = self.parse_block()?;

    let else_block = if self.peek().kind == TokenKind::Else {
        self.bump();
        self.parse_block()?
    } else {
        Vec::new()
    };

    Ok(Stmt::If {
        cond,
        then_block,
        else_block,
    })
}

fn parse_while(&mut self) -> Result<Stmt, ParseError> {
    self.expect(TokenKind::While)?;
    let cond = self.parse_expr()?;
    let body = self.parse_block()?;

    Ok(Stmt::While { cond, body })
}

fn parse_expr_stmt(&mut self) -> Result<Stmt, ParseError> {
    let expr = self.parse_expr()?;
    self.expect(TokenKind::Semicolon)?;
```

```
        Ok(Stmt::Expr(expr))
    }

    fn parse_expr(&mut self) -> Result<Expr, ParseError> {
        self.parse_binary_expr(0)
    }

    // Precedence-climbing / Pratt-style binary expression parser
    // min_prec: minimum precedence to accept in this call
    fn parse_binary_expr(&mut self, min_prec: u8) -> Result<Expr, ParseError> {
        // 1. parse left-hand side primary
        let mut lhs = self.parse_primary()?;

        loop {
            // 2. check if next token is a binary operator and get its precedence
            let (op, prec) = match self.token_to_binop_prec(&self.peek().kind) {
                Some(x) => x,
                None => break,
            };

            if prec < min_prec {
                break;
            }

            // 3. consume operator
            self.bump();

            // 4. parse RHS with higher precedence (left-associative operators use
prec+1)
            let next_min = prec + 1;
            let rhs = self.parse_binary_expr(next_min)?;

            // 5. build binary node and continue
            lhs = Expr::Binary {
                op,
                left: Box::new(lhs),
                right: Box::new(rhs),
            };
        }

        Ok(lhs)
    }

    // Map a token kind to (BinOp, precedence). Higher number = higher precedence.
    // Precedence levels (high -> low): * / = 4, + - = 3, < > <= >= = 2, == != = 1
    fn token_to_binop_prec(&self, kind: &TokenKind) -> Option<(BinOp, u8)> {
        match kind {
            TokenKind::Star => Some((BinOp::Mul, 4)),
            TokenKind::Slash => Some((BinOp::Div, 4)),
            TokenKind::Plus => Some((BinOp::Add, 3)),
            TokenKind::Minus => Some((BinOp::Sub, 3)),
            TokenKind::Less => Some((BinOp::Lt, 2)),
            TokenKind::LessEqual => Some((BinOp::Le, 2)),
            TokenKind::Greater => Some((BinOp::Gt, 2)),
            TokenKind::GreaterEqual => Some((BinOp::Ge, 2)),
            TokenKind::EqualEqual => Some((BinOp::Eq, 1)),
```

```
            TokenKind::NotEqual => Some((BinOp::NotEq, 1)),
            _ => None,
        }
    }

    fn parse_primary(&mut self) -> Result<Expr, ParseError> {
        let tok = self.bump().clone();

        match tok.kind {
            TokenKind::Int(value) => Ok(Expr::Int(value)),

            TokenKind::Ident(name) => {
                if self.peek().kind == TokenKind::LParen {
                    self.bump(); // '('
                    let mut args = Vec::new();

                    if self.peek().kind != TokenKind::RParen {
                        loop {
                            args.push(self.parse_expr()?);

                            if self.peek().kind == TokenKind::Comma {
                                self.bump();
                            } else {
                                break;
                            }
                        }
                    }

                    self.expect(TokenKind::RParen)?;
                    Ok(Expr::Call { name, args })
                } else {
                    Ok(Expr::Local(name))
                }
            }

            TokenKind::LParen => {
                let expr = self.parse_expr()?;
                self.expect(TokenKind::RParen)?;
                Ok(expr)
            }
            _ => Err(ParseError::ExpectedExpression { span: tok.span }),
        }
    }
}

#[allow(dead_code)]
#[derive(Debug)]
pub enum ParseError {
    UnexpectedToken { expected: String, found: Token },
    ExpectedExpression { span: Span },
}
```

7.0.6 src/runner.rs

```
use std::sync::{Arc, Mutex};
```

```rust
use wasmtime::{Engine, Instance, Store, Val};

use crate::codegen::module::ModuleGen;
use crate::lexer::lex_file;
use crate::parser::Parser;

/// Compile source to wasm bytes.
pub fn compile_bytes_from_src(src: &str) -> Result<Vec<u8>, String> {
    let tokens = lex_file(src).map_err(|e| format!("lex error: {:?}", e))?;
    let mut parser = Parser::new(&tokens);
    let program = parser
        .parse_program()
        .map_err(|e| format!("parse error: {:?}", e))?;

    // Register host imports (print), then declare and emit all functions.
    let mut module_gen = ModuleGen::new().init_with_host_functions();
    for func in &program.functions {
        module_gen.declare_function(func);
    }
    for func in &program.functions {
        module_gen.emit_function(func);
    }
    let bytes = module_gen.finish();
    Ok(bytes)
}

/// Run wasm bytes calling `main` with the provided i64 arguments.
/// Returns Ok(Some(i64)) if the function returns a single i64, Ok(None) if
/// the function has no return, or Err on failure.
pub fn run_wasm_bytes(bytes: &[u8], args: Vec<i64>) -> Result<Option<i64>, String>
{
    let engine = Engine::default();
    let module = wasmtime::Module::from_binary(&engine, bytes)
        .map_err(|e| format!("module compile error: {}", e))?;

    let mut store = Store::new(&engine, ());

    // create host print function
    let print_func = wasmtime::Func::wrap(&mut store, |v: i64| {
        println!("{}", v);
    });

    let instance = Instance::new(&mut store, &module, &[print_func.into()])
        .map_err(|e| format!("instance error: {}", e))?;

    let func = instance
        .get_func(&mut store, "main")
        .ok_or_else(|| "function `main` not found".to_string())?;

    // Prepare Val parameters
    let params: Vec<Val> = args.into_iter().map(Val::I64).collect();

    // Inspect function type to determine result count
    let ty = func.ty(&store);
    let results = ty.results().len();
```

```
    let mut results_buf: Vec<Val> = vec![Val::I64(0); results];

    func.call(&mut store, &params, &mut results_buf)
        .map_err(|e| format!("runtime error: {}", e))?;

    if results == 1 {
        if let Val::I64(v) = results_buf[0] {
            Ok(Some(v))
        } else {
            Err("unexpected return value type".to_string())
        }
    } else {
        Ok(None)
    }
}

/// Convenience helper: compile source and run it.
pub fn run_source(src: &str, args: Vec<i64>) -> Result<Option<i64>, String> {
    let bytes = compile_bytes_from_src(src)?;
    run_wasm_bytes(&bytes, args)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn run_if_gt_sample() {
        let src = "fn main(x, y) -> Int { if (x > 5) { return 1; } else { return
0; } }";
        let res = run_source(src, vec![64, 8]).expect("run failed");
        assert_eq!(res, Some(1));
    }

    #[test]
    fn run_add_compare() {
        let src = "fn main(a, b) -> Int { let z = a + 1 < b; return z; }";
        // a=1, b=3 -> 1+1 < 3 -> true -> 1
        let res = run_source(src, vec![1, 3]).expect("run failed");
        assert_eq!(res, Some(1));
    }

    #[test]
    fn run_parenthesized_complex() {
        let src = "fn main(a,b,c,d) -> Int { let r = (a + b) <= (c - d); return
r; }";
        // Wrong arity should return an error.
        let _res =
            run_source(src, vec![1, 1]).expect_err("expected error because missing
c,d args");

        let src2 = "fn main(a,b,c,d) -> Int { let r = (a + b) <= (c - d); return
r; }";
        let bytes = compile_bytes_from_src(src2).expect("compile failed");
        let out =
```

```
            run_wasm_bytes(&bytes, vec![2, 1]).expect_err("expected error due to
wrong arg count");
        let _ = out;
    }

    #[test]
    fn print_statement_outputs() {
        // inline source instead of reading a file
        let src = "fn main(){ print(3); return; }";

        let bytes = compile_bytes_from_src(src).expect("compile failed");

        // instantiate module with a host `print_i64` that records values
        let engine = Engine::default();
        let module = wasmtime::Module::from_binary(&engine, &bytes).expect("module
compile failed");

        let printed: Arc<Mutex<Vec<i64>>> = Arc::new(Mutex::new(Vec::new()));

        let mut store = Store::new(&engine, ());
        let captured = printed.clone();
        let print_func = wasmtime::Func::wrap(&mut store, move |v: i64| {
            captured.lock().unwrap().push(v);
        });

        let instance = Instance::new(&mut store, &module, &[print_func.into()])
            .expect("instance creation failed");

        let func = instance
            .get_func(&mut store, "main")
            .expect("main not found");
        let params: Vec<Val> = Vec::new();
        let mut results_buf: Vec<Val> = Vec::new();

        func.call(&mut store, &params, &mut results_buf)
            .expect("call failed");

        let got = printed.lock().unwrap().clone();
        assert_eq!(got, vec![3]);
    }
}
```

7.0.7 src/token.rs

```
#[derive(Debug, Clone, PartialEq)]
pub enum TokenKind {
    // Keywords
    Let,
    Fn,
    If,
    Else,
    While,
    Return,

    // Identifiers + literals
```

```rust
    Ident(String),
    Int(i64),

    // Operators
    Plus,
    Minus,
    Star,
    Slash,
    Percentage,
    Equal,

    // Comparison operators
    EqualEqual,
    NotEqual,
    Less,
    LessEqual,
    Greater,
    GreaterEqual,

    // Delimiters
    LParen,
    RParen,
    LBrace,
    RBrace,
    Semicolon,
    Comma,

    // Type hints
    Colon,
    Arrow,
    IntType,

    EOF,
}

impl TokenKind {
    pub fn name(&self) -> String {
        match self {
            TokenKind::Ident(_) => "identifier".to_string(),
            TokenKind::Int(_) => "integer literal".to_string(),
            _ => format!("{:?}", self),
        }
    }
}

#[derive(Debug, Clone)]
pub struct Token {
    pub kind: TokenKind,
    pub span: Span,
}

#[derive(Debug, Clone)]
pub struct Span {
    pub start: usize,
    pub end: usize,
}
```

7.0.8 src/codegen/expr.rs

```
use std::collections::HashMap;

use crate::{
    ast::{BinOp, Expr},
    codegen::ir::IrInstruction,
    codegen::module::FuncGen,
};

pub fn emit_expr(expr: &Expr, r#gen: &mut FuncGen, funcs: &HashMap<String, (u32,
bool)>) -> bool {
    match expr {
        Expr::Int(v) => {
            r#gen.instructions.push(IrInstruction::I64Const(*v));
            true
        }
        Expr::Local(name) => {
            let idx = r#gen.local_map[name];
            r#gen.instructions.push(IrInstruction::LocalGet(idx));
            true
        }
        Expr::Binary { op, left, right } => {
            let _ = emit_expr(left, r#gen, funcs);
            let _ = emit_expr(right, r#gen, funcs);

            match op {
                BinOp::Add => r#gen.instructions.push(IrInstruction::I64Add),
                BinOp::Sub => r#gen.instructions.push(IrInstruction::I64Sub),
                BinOp::Mul => r#gen.instructions.push(IrInstruction::I64Mul),
                BinOp::Div => r#gen.instructions.push(IrInstruction::I64DivS),
                BinOp::Eq => {
                    r#gen.instructions.push(IrInstruction::I64Eq);
                    // Keep expression results as i64 across the compiler.
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                BinOp::Lt => {
                    r#gen.instructions.push(IrInstruction::I64LtS);
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                BinOp::Gt => {
                    r#gen.instructions.push(IrInstruction::I64GtS);
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                // a != b  =>  !(a == b)
                BinOp::NotEq => {
                    r#gen.instructions.push(IrInstruction::I64Eq);
                    r#gen.instructions.push(IrInstruction::I32Eqz);
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                // a <= b  =>  !(a > b)
                BinOp::Le => {
                    r#gen.instructions.push(IrInstruction::I64GtS);
                    r#gen.instructions.push(IrInstruction::I32Eqz);
```

```
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
                // a >= b  =>  !(a < b)
                BinOp::Ge => {
                    r#gen.instructions.push(IrInstruction::I64LtS);
                    r#gen.instructions.push(IrInstruction::I32Eqz);
                    r#gen.instructions.push(IrInstruction::I64ExtendI32S);
                }
            }

            true
        }
        Expr::Call { name, args } => {
            for arg in args {
                let _ = emit_expr(arg, r#gen, funcs);
            }
            let (idx, has_ret) = funcs[name];
            r#gen.instructions.push(IrInstruction::Call(idx));
            has_ret
        }
    }
}
```

7.0.9 src/codegen/ir.rs

```
use wasm_encoder::{BlockType, Instruction};

pub enum IrInstruction {
    I64Const(i64),
    I64Eqz,
    BrIf(u32),
    Br(u32),
    LocalSet(u32),
    LocalGet(u32),
    Call(u32),
    If(BlockType),
    Else,
    Block(BlockType),
    Loop(BlockType),
    Drop,
    Return,
    End,

    // Arithmetic
    I64Add,
    I64Sub,
    I64Mul,
    I64DivS,
    I64Eq,
    I64LtS,
    I64GtS,
    I64ExtendI32S,
    I32Eqz,
}
```

```
impl IrInstruction {
    pub fn to_wasm(&self) -> Instruction<'_> {
        match self {
            IrInstruction::I64Const(v) => Instruction::I64Const(*v),
            IrInstruction::I64Eqz => Instruction::I64Eqz,
            IrInstruction::BrIf(idx) => Instruction::BrIf(*idx),
            IrInstruction::Br(idx) => Instruction::Br(*idx),
            IrInstruction::LocalSet(idx) => Instruction::LocalSet(*idx),
            IrInstruction::LocalGet(idx) => Instruction::LocalGet(*idx),
            IrInstruction::Call(idx) => Instruction::Call(*idx),
            IrInstruction::If(block_type) => Instruction::If(*block_type),
            IrInstruction::Else => Instruction::Else,
            IrInstruction::Block(block_type) => Instruction::Block(*block_type),
            IrInstruction::Loop(block_type) => Instruction::Loop(*block_type),
            IrInstruction::Drop => Instruction::Drop,
            IrInstruction::Return => Instruction::Return,
            IrInstruction::End => Instruction::End,
            IrInstruction::I64Add => Instruction::I64Add,
            IrInstruction::I64Sub => Instruction::I64Sub,
            IrInstruction::I64Mul => Instruction::I64Mul,
            IrInstruction::I64DivS => Instruction::I64DivS,
            IrInstruction::I64Eq => Instruction::I64Eq,
            IrInstruction::I64LtS => Instruction::I64LtS,
            IrInstruction::I64GtS => Instruction::I64GtS,
            IrInstruction::I64ExtendI32S => Instruction::I64ExtendI32S,
            IrInstruction::I32Eqz => Instruction::I32Eqz,
        }
    }
}
```

7.0.10 src/codegen/mod.rs

```
mod expr;
mod ir;
pub mod module;
mod stmt;
```

7.0.11 src/codegen/module.rs

```
use std::collections::HashMap;

use crate::codegen::ir::IrInstruction;
use wasm_encoder::*;

use crate::{ast::FunctionDecl, codegen::stmt::emit_stmt};

pub struct ModuleGen {
    module: Module,
    types: TypeSection,
    imports: ImportSection,
    functions: FunctionSection,
    codes: CodeSection,
    exports: ExportSection,

    func_indices: HashMap<String, (u32, bool)>,
```

```
    next_type_index: u32,
    next_func_index: u32,
}

impl ModuleGen {
    pub fn new() -> Self {
        Self {
            module: Module::new(),
            types: TypeSection::new(),
            imports: ImportSection::new(),
            functions: FunctionSection::new(),
            codes: CodeSection::new(),
            exports: ExportSection::new(),
            func_indices: HashMap::new(),
            next_type_index: 0,
            next_func_index: 0,
        }
    }

    pub fn init_with_host_functions(mut self) -> Self {
        // Register host imports (currently only print).
        self.add_print_import();
        self
    }

    fn add_print_import(&mut self) {
        // (i64) -> ()
        let type_index = self.next_type_index;
        self.next_type_index += 1;
        self.types.ty().function(vec![ValType::I64], Vec::new());

        // Import as env.print_i64.
        self.imports
            .import("env", "print_i64", EntityType::Function(type_index));

        // Expose this import as `print` in the compiler function table.
        let idx = self.next_func_index;
        self.next_func_index += 1;
        self.func_indices.insert("print".to_string(), (idx, false));
    }

    pub fn finish(mut self) -> Vec<u8> {
        self.module.section(&self.types);
        self.module.section(&self.imports);
        self.module.section(&self.functions);
        self.module.section(&self.exports);
        self.module.section(&self.codes);
        self.module.finish()
    }

    pub fn declare_function(&mut self, func: &FunctionDecl) {
        let type_index = self.next_type_index;
        self.next_type_index += 1;

        let params = vec![ValType::I64; func.params.len()];
        let results = if func.return_type.is_some() {
```

```
            vec![ValType::I64]
        } else {
            Vec::new()
        };

        self.types.ty().function(params, results);

        let idx = self.func_indices.len() as u32;
        self.func_indices
            .insert(func.name.clone(), (idx, func.return_type.is_some()));

        self.functions.function(type_index);
        self.exports.export(&func.name, ExportKind::Func, idx);
    }

    pub fn emit_function(&mut self, func: &FunctionDecl) {
        let mut r#gen = FuncGen {
            locals: Vec::new(),
            local_map: HashMap::new(),
            instructions: Vec::new(),
            has_return: func.return_type.is_some(),
        };

        for (i, name) in func.params.iter().enumerate() {
            r#gen.local_map.insert(name.clone(), i as u32);
        }

        for stmt in &func.body {
            emit_stmt(stmt, &mut r#gen, &self.func_indices);
        }

        // If the function has a return type but no explicit return was emitted,
        // provide a default 0 return so the function type matches. If no return
        // type is declared, don't emit a return value.
        if func.return_type.is_some() {
            r#gen.instructions.push(IrInstruction::I64Const(0));
            r#gen.instructions.push(IrInstruction::Return);
        }

        let mut local_groups = Vec::new();
        for ty in r#gen.locals {
            local_groups.push((1, ty));
        }

        let mut wasm_func = Function::new(local_groups);

        for instr in r#gen.instructions {
            wasm_func.instruction(&instr.to_wasm());
        }

        wasm_func.instruction(&Instruction::End);

        self.codes.function(&wasm_func);
    }
}
```

```rust
pub struct FuncGen {
    pub locals: Vec<ValType>,
    pub local_map: HashMap<String, u32>,
    // Instruction list for one function body.
    pub instructions: Vec<IrInstruction>,
    pub has_return: bool,
}
```

7.0.12 src/codegen/stmt.rs

```rust
use crate::codegen::ir::IrInstruction;
use crate::{
    ast::Stmt,
    codegen::{expr::emit_expr, module::FuncGen},
};
use std::collections::HashMap;
use wasm_encoder::*;

pub fn emit_stmt(stmt: &Stmt, r#gen: &mut FuncGen, funcs: &HashMap<String, (u32,
bool)>) {
    match stmt {
        Stmt::Let { name, value } => {
            let idx = r#gen.local_map.len() as u32;
            r#gen.locals.push(wasm_encoder::ValType::I64);
            r#gen.local_map.insert(name.clone(), idx);

            emit_expr(value, r#gen, funcs);
            r#gen.instructions.push(IrInstruction::LocalSet(idx));
        }

        Stmt::Expr(expr) => {
            // Drop only if this expression produced a stack value.
            let produced = emit_expr(expr, r#gen, funcs);
            if produced {
                r#gen.instructions.push(IrInstruction::Drop);
            }
        }

        Stmt::Return(expr_opt) => {
            match expr_opt {
                Some(expr) => {
                    emit_expr(expr, r#gen, funcs);
                }
                None => {
                    // If a return type exists, return a default zero value.
                    if r#gen.has_return {
                        r#gen.instructions.push(IrInstruction::I64Const(0));
                    }
                }
            }

            r#gen.instructions.push(IrInstruction::Return);
        }

        Stmt::If {
```

```
                cond,
                then_block,
                else_block,
            } => {
                emit_expr(cond, r#gen, funcs);
                // emit_expr leaves i64 values, while wasm `if` expects i32.
                // Convert i64 truthy/falsey into an i32 condition.
                r#gen.instructions.push(IrInstruction::I64Eqz);
                r#gen.instructions.push(IrInstruction::I32Eqz);
                r#gen.instructions.push(IrInstruction::If(BlockType::Empty));

                for s in then_block {
                    emit_stmt(s, r#gen, funcs);
                }

                if !else_block.is_empty() {
                    r#gen.instructions.push(IrInstruction::Else);
                    for s in else_block {
                        emit_stmt(s, r#gen, funcs);
                    }
                }

                r#gen.instructions.push(IrInstruction::End);
            }

            Stmt::While { cond, body } => {
                r#gen
                    .instructions
                    .push(IrInstruction::Block(BlockType::Empty));
                r#gen
                    .instructions
                    .push(IrInstruction::Loop(BlockType::Empty));

                emit_expr(cond, r#gen, funcs);
                r#gen.instructions.push(IrInstruction::I64Eqz);
                r#gen.instructions.push(IrInstruction::BrIf(1));

                for s in body {
                    emit_stmt(s, r#gen, funcs);
                }

                r#gen.instructions.push(IrInstruction::Br(0));
                r#gen.instructions.push(IrInstruction::End);
                r#gen.instructions.push(IrInstruction::End);
            }
        }
    }
}
```